QwyitCipher™ (QCy™)
Qwyit® PDAF_SEC Cipher Configuration

Reference Guide

Version 2.0 Mar 11, 2021

Copyright Notice

Copyright © 2021 Qwyit LLC. All Rights Reserved.

Abstract

This paper provides a technical overview of the QwyitCipher™ (QCy™). The Qwyit protocol is built on several distinct, innovative primitives. These include MOD16, Combine, Extract, PDAF and OWC functions. All of these in different combinations can create perfectly secure (underdetermined) Qwyit™ ciphers, with the simplest using the PDAF in a one-step encrypt/decrypt function (PDAF_SEC). Message traffic is authenticated and encrypted. The cipher documented here is not the only possible one; it's the simplest and the world's fastest and most secure encryption engine.

Contents

QwyitCipher™ PDAF_SEC Configuration (QCy™) – Reference Guide

This document outlines the simplest QwyitCipher™ (QCy™) configuration, based on the Qwyit® protocol primitives. These include MOD16, Combine, Extract, PDAF and OWC functions. The configuration detailed here is certainly not the only possible one, but it is the world's simplest, fastest most secure encryption engine. Wherever the term QCy™ is used in any Qwyit® document, it refers to any configuration, and specifically the PDAF_SEC one listed here. Implementation is based on the full Qwyit® protocol as outlined in the current version of the *Qwyit Protocol Reference* document, available from Qwyit LLC. QCy™ client demo APIs are available from Qwyit LLC; go to www.qwyit.com.

Introduction

The Qwyit™ protocol provides authentication (embedded) and data security (stream cipher) for digital communications, assets and networks using a secret-key system. Many QCy™ (Q-Sigh) stream cipher configurations can be built using different combinations of the Qwyit® primitives: MOD16 and MOD16D, Combine and Extract, PDAF and OWC. The exact definitions and inner workings of the functions are found in the *Qwyit Protocol Reference*. The new PDAF_SEC cipher function can be thought of as another Qwyit™ primitive, as it is a single-step encryption function. The function and reference code are specified here.

Approach

Qwyit® is a Security Engineering company. We're not cryptographers, more like Information Theorists. Our protocol, Qwyit™, and QCy™ cipher described here, are presented as real world solutions to the fundamental flaws and lack of universal, easy-to-use and properly applicable privacy and security in digital communications, storage and not-present transactions.

Instead of the minutiae of cryptographic math proofs that attempt to categorize authentication and data security methods – because this does absolutely nothing to help digital architects, owners and users understand, properly apply and implement them – there is a straightforward engineering check for a provably secure methodology: the Perfect Security Cross test. This is a form of *complete cryptanalysis* – if an algorithm provides all four barriers in the cross, the method/algorithm/system is Perfectly Secret.

That term, *Perfect Secrecy*, has been applied by cryptography beginning with Shannon's 1949 landmark paper in which he defined and provided mathematic proof that such a thing can be only *one* thing. The problem with current cryptography, is the rest of Shannon's paper has been unheeded. Here are two crucial direct quotes about *proving* security:

"It is difficult to define the pertinent ideas involved with sufficient precision to obtain results in the form of mathematical theorems, but it is believed that the conclusions, in the form of general principles, are correct."

This quote is from Shannon's *Communication Theory of Secrecy Systems**, near the end of Part III *Practical Secrecy,* Section 21 *The Work Characteristic*, where he's about to embark on discussion in solving this question that he just proposed:

"How can we ever be sure that a system which is not ideal and therefore *has* a unique solution for sufficiently large *N* will require a large amount of work to break with *every* method of analysis?"

The italics are his, and they emphasize that for cryptographic solutions that are "not ideal", he's asking, and pointing out, that one can't really *prove* that these non-ideal systems always work. What he's talking about – cryptosystems that aren't "ideal" – *these are every single one of today's cryptographic algorithms!*

What happened in cryptography with "Ideal Systems"? Everyone today is well versed in his *Perfect Secrecy*, the proof, and the definition (short version): Key as long as the message. And *only* that. But…Shannon actually stated – and detailed – another definition of a provably secure system (our red accent, his italics):

"It is possible to construct secrecy systems with a finite key for certain "languages" in which the equivocation does not approach zero as $N \to \infty$. In this case, no matter how much material is intercepted, the enemy still does not obtain a unique solution to the cipher but is left with many alternatives, all of reasonable probability. Such systems we call *ideal* systems. It is possible in any language to approximate such behavior—i.e., to make the approach to zero of $H(N)$ recede out to arbitrarily large *N*. However, such systems have a number of drawbacks, such as complexity and sensitivity to errors in transmission of the cryptogram."

In that last sentence, as Shannon goes on to construct an example *Ideal System*, he notes the difficulty; but his focus was on 1949 'networks', which were only text and languages. Computing, bits and digital 'languages' were a decade away. So his "complexity" revolved around them; and he ended his Part II, *Theoretical Secrecy* section stating:

"Ideal secrecy systems suffer from a number of disadvantages."

Then, off he went to Part III where we started above, the non-ideal systems, and…that's where every single cryptographer since has been focused (stuck, really.) And what a shame the industry has wasted 70+ years working in the wrong chapter. The result?

- End-to-end Security isn't achievable because the security systems aren't
- Constant attempts to 'balance' Security vs Performance, when we should have them *both*
- Systems that no one understands – developers, operators, users
- Our Status Quo 'Best Practices' give up $6*trillion yearly* in cybercrime

The missing ingredient in all of these is that there hasn't been any properly applied *engineering* of Perfect Secrecy cryptography into real world network solutions to realize Shannon's IDEAL SYSTEM

Everyone has been working in the wrong chapter but us: Qwyit has picked right up where Shannon left off, and we've actually designed, built and tested a *Perfect Secrecy Ideal System*.

QCy™ incorporates our engineering innovations and delivers the fastest possible, most efficient, provably secure digital data cipher. It is built with simple innovative cryptographic primitives that provide a true perfectly secure, endless key, *Authentic Encryption*[Note1] cipher – security discussion in Appendices D and F. The QCy™ PDAF_SEC implementation meets the key requirement of Perfectly Secure new key bit per plaintext bit and delivers the ultimate solution to the 'Perfect Security Cross'.

QCy™ creates Shannon’s *Perfect Secrecy Ideal System* and delivers Perfect Cross SecurityNote2:

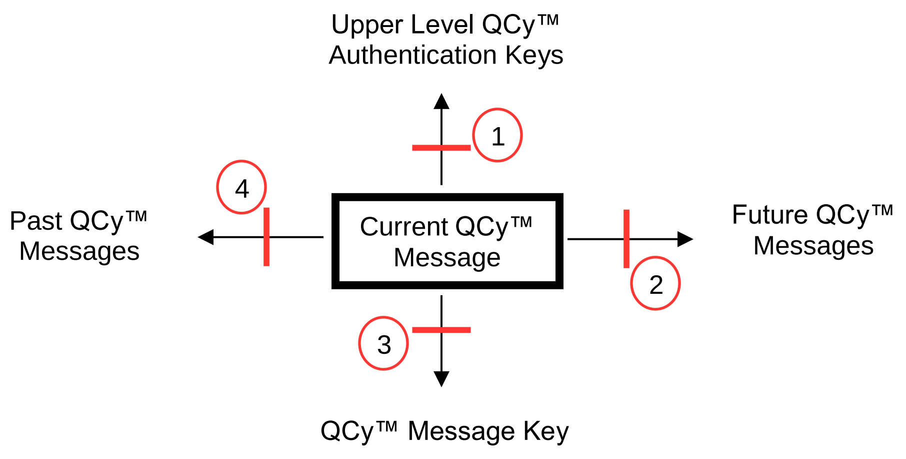
QCy™ Perfect Security Cross

There is a multitude of confusing cryptographic features, properties, ‘proofs’ and definitions that are purported to provide attack limitations. But there are only two security questions that matter:

1. Does the system use a proven *Perfect Secrecy* cipher? (Either an OTP or an Ideal System, since these are the only ones; every other cipher is broken)
2. Is there continual, mutual Authentication of every digital interaction? (If there isn’t, then all attacks are possible)

Once those two questions are answered satisfactorily (both *Yes*), then there is a straightforward cryptanalysis that will fundamentally determine system perfection:

QCy™ Perfect Security Cross



System Assumptions:

- Upper Level QCy™ Authentication Keys are securely pre-shared
  - By participant-managed, independent trust QwyitKey™ system, or other
- QCy™ Auth Keys create unique, new message keys for every message

Perfect Security IF:

1. Broken Current Message does NOT reveal Authentication Keys (One-way math gate)
2. Broken Current Message does NOT reveal Future Messages (One-way math gate)
3. Current Message is Provably Secure (Math proved, Shannon Secure)
   a. Known Plaintext reveals key, but 1, 2 and 4 still hold – as does any remaining msg
4. Broken Current Message does NOT reveal Past Messages (One-way math gate)

The system is Perfect Cross Secure when it stops all attacks in all directions

QCy™ PDAF_SEC Overview

The Qwyit™ Authentication and Data Security protocol includes a unique and property-filled function called the *Position Digit Algebra Function (PDAF)*. More information can be found in the *Qwyit™ Protocol Reference Guide* and reference code is included in Appendix E. To use the function as a simple, one-step underdetermined cipher, the PDAF becomes the PDAF_SEC function by minimally adding to it to include performing the one-step simple XOR of the PDAF result with the TargetText, either plaintext to be encrypted or ciphertext to be decrypted. The process of the PDAF_SEC:

- Requires, ideally, two input *n*-bit (256 currently) key values (or one that will function as both keys – not recommended as this reduces the initial key space); calling one the *ValueKey* (*VK*) and the other the *OffsetKey* (*OK*). An **OpenReturn** (**OR**), which is a same-bit-sized public Initialization Vector, is also included; as well as either the plaintext (*PT*) to be encrypted or the ciphertext (**CT**) to be decrypted
  - Both keys should be the same number of bits (digits) – although this is not required (the function can be written to accommodate short pointing keys that are concatenated to meet the return requirements, although this is not recommended)
  - Begin by MOD adding the **OR** and the Qwyit Authentication Key *QK* to create a unique starting *ValueKey* pointer VK^P, then perform a PDAF function of that pointer VK^P with the Qwyit Authentication key *EK*. This creates an underdetermined starting *ValueKey* that even if discovered/stolen won't leak the Master Qwyit system Authentication Keys
    - The Master Qwyit Auth Keys should be stored/managed separately from the QCy™ encryption keys
  - Create the starting *OffsetKey* in the same manner, by MOD16 adding the **OR** and the Qwyit Authentication Key *EK* to create a unique starting *OffsetKey* pointer OK^P, then perform a PDAF function of that pointer OK^P with the Qwyit Authentication key *QK*
  - *NOTE*: Because QCy™ is designed to be one-step, extremely flexible for implementation everywhere, *the key lengths can be any size*. There are no block requirements, no restrictions on size. Recommendation is for an even number of bits for simplicity

- Set pointers at digit position 1 in each key (*Po* in the *OK*, and *Pv* in the *VK*)
  - Digit 'position' can be any bit-length best and most easily implemented; e.g., 4-bits, 8-bit bytes, entire 256-bit blocks, etc. The following digit position rules apply to whatever is being implemented. This document details examples in 4-bit digit positions

- Take the value at *Pv* and MOD16 add it to the digit in the *Po* position
  - E.g., if the *VK* is "19B3AD2" and the *OK* is "C0FF48C", the first PDAF result is 1 + C = D where the 1 is from the *Pv* at position one, and the C is selected from the *Po* in the *OK*
  - These results in total form the Message Key, *W*
    - *NOTE:* See the following Selection Options section for additional ways to perform the PDAF on *VK/OK*. All of the methods produce underdetermined message keys

- Take the PDAF add result (*W*) and XOR it with the TargetText (*PT* or **CT**) first 'value'
  - E.g., if the TargetText is *PT* = "This is a test.", the PDAF select add result of D is XOR'd with "T"
    - The TargetText can be 'value' sliced any way, in 4 or 8-bit 'values', etc.; as long as the XOR of key bit and TargetText bit is performed uniquely and non-repetitively

- Move both pointers one position to the right
    - A complete cycle is from *1..n* digits in the *VK* and *OK*

- If there is more TargetText remaining at the end of the cycle (denoted by pointer *Pc*), increment *Pc* by one, move the *Pv* pointer one digit position to the right of its last cycle start (e.g., the 2nd cycle will start at digit position 2, 3rd cycle at position 3, etc.), and perform another same select add/cipher XOR cycle where the *Po* always starts at the first *OK* position
    - There will be Length(*VK*) squared cycles in total. E.g., a 256-bit key, containing 64 4-bit hex values, will have 64^2 (4,096) returned PDAF_SEC 4-bit digit results
        - This is for each Selection Cycle Case noted below; using all three Selection Cycle Cases will yield 64^2 x 3 (12,288) returned PDAF_SEC 4-bit digit results

- If there is more TargetText remaining at the end of *all* cycles (a complete cycle set), update the *VK* and *OK*, reset the *Pv* and *Po* to the 1st position and continue as above until the end of the plaintext/ciphertext (at the end of each cycle set, continue updating/cycling until PT/CT is exhausted)
    - The update must reconfigure both the *VK* and *OK* such that there is a mathematic one-way gate 'across' the cycling resets. If it's possible to write a single, continuing equation where VK^{New} and OK^{New} are known extensions of VK^{Start} and OK^{Start}, then any positively known (impossibly broken, but possibly known) Plaintext would result in the continued ability to derive *W* and realize the correct message content (the key values wouldn't necessarily be known if they are combined in some fashion, but their result would be)
    - The QCy™ one-way gate is the Qwyit PDAF function. It provides a one-way linear MOD16 across digit positions, by mixing/adding different positions of both keys pointing 'into' each other as dictated by the values in the keys. See Appendix E for function definition and two operating modes (Offset Key Add and Dual Key Add)
    - In order to quickly reconfigure both keys, perform a PDAF using VK^P to point into *OK* to update *VK* and another PDAF using OK^P to point into *VK* to update *OK*. This reconfiguration provides the *Random Rearrangement* property of maintaining a new, unique key bit for every plaintext bit. See Appendix D and F for complete *Random Rearrangement* definition and security explanation of this innovative cryptographic function
    - Then continue the next cycle set with the reconfigured keys
        - This Update Method, *Random Rearrangement* value mixing, creates a never-ending series of new keys that are distinctly unique, always maintaining their random entropy and that create the required endless, never-repeating child message keys for encryption. This property of *Random Rearrangement* delivers a *true Ideal System*

Default QCy™ PDAF_SEC Cipher

The PDAF_SEC's process of underdetermined linear equations:

Qwyit™ ID:	**OpenID**	[This is the public Qwyit™ Community ID]
Qwyit™ Keys:	*EK*, *QK*	[These are the upper level Qwyit™ Authentication Keys]
Initialization Vector:	**OR**	[A randomly generated public Initialization Vector]

Session Start: QK MOD16 **OR** = VK^P then PDAF(EK, VK^P) = VK^C [Starting *ValueKey*]
EK MOD16 **OR** = OK^P then PDAF(QK, OK^P) = OK^C [Starting *OffsetKey*]

Selection: $VK^{Pc[1\ldots n]}{}_{Pv[1\ldots n]}$ MOD16 $OK_{Po[1\ldots n]} = W_{1\ldots n^2}$
Where pointer *Pv* and *Po* increment +1 through the key length for each cycle pointer *Pc* [This is a PDAF in Dual Key mode]
If repeating, substitute VK^N and OK^N for each new cycle

Cipher: $W_{1\ldots n^2} \oplus PT_{1\ldots n^2} = \mathbf{CT}_{1\ldots n^2}$ repeating w/next cycle selection if more *PT*

Update (more *PT*): Update ValueKey: PDAF(OK^C, VK^P) = VK^{Next}
Update OffsetKey: PDAF(VK^C, OK^P) = OK^{Next}
Where the PDAF performed is the Key Offset Add mode

Repeat: Cycle through Selection, Cipher, Updates, replacing *VK* and *OK* until *PT*, **CT** completed

Send: Per Message[**OpenID**, **OR**, **CT**] to the **OpenID** location of intended recipient

QCy™ PDAF_SEC Selection Options

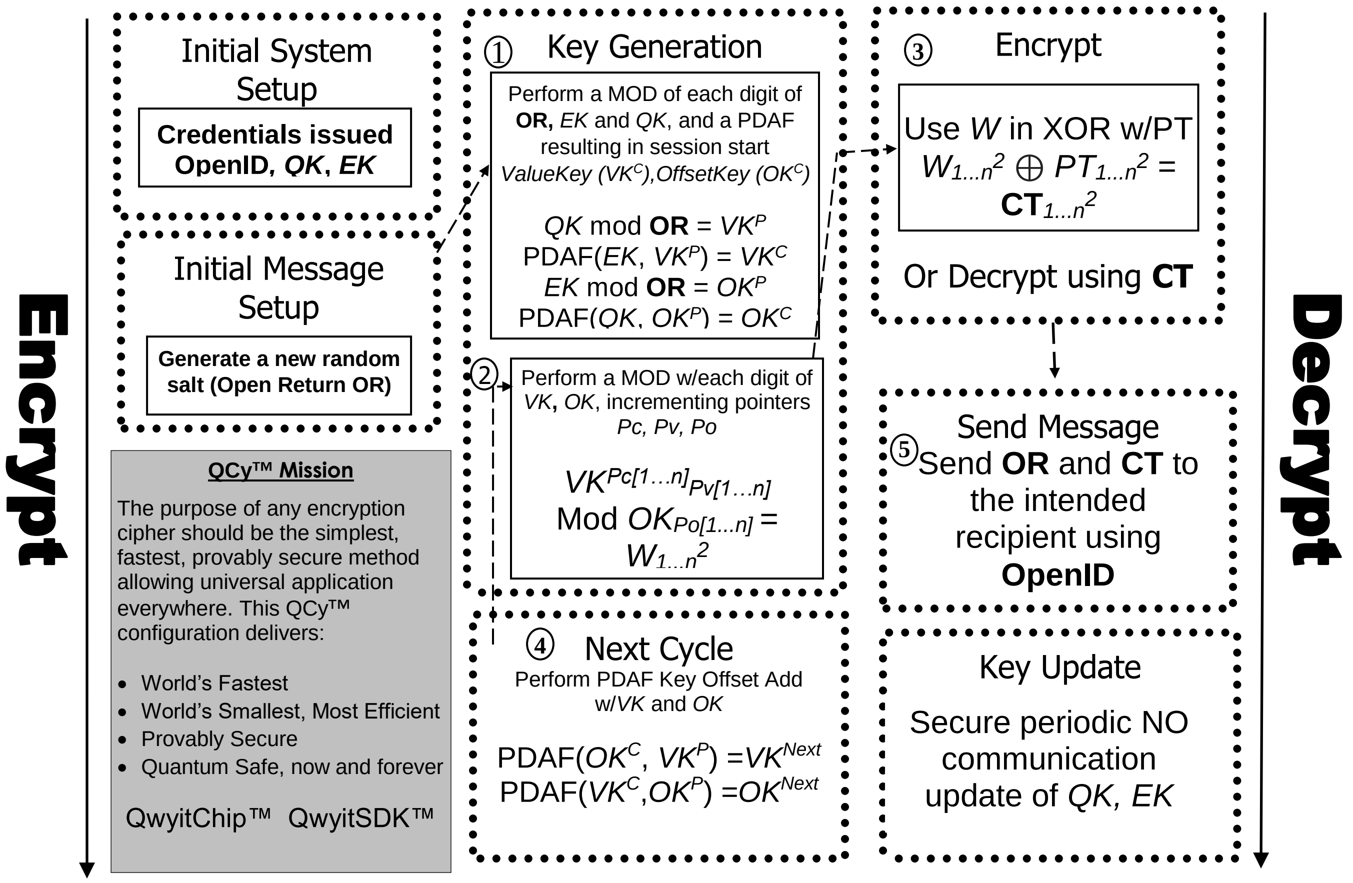
In Appendices D and F, the Perfect Secrecy provided by QCy™ is detailed; the irrefutable mathematic foundation for this security is that QCy™ is underdetermined at every step. In that regard, there are additional options for the Selection step. This step provides the endless key values using a PDAF mode – either an Offset Key Add mode or the Dual Key Add mode. The default case shown above, uses the PDAF Dual Key Add mode. *Any of these can be used, in any combination, as they all remain underdetermined and provide unique results*. The choice can be made based on performance, code space, etc. These additional Selection Case methods are outlined in the Reference Code in Appendix E:

Case	General Equation*	Description
1	VK^1 mod $OK^1 = W^1$	Default: DUAL KEY ADD mode, ValueKey plus OffsetKey, starting at digit 1 of each and adding that position. Continue until all digits have been added to each other
2	VK^1 mod $VK^X = W^1$	OFFSET KEY ADD mode, ValueKey plus OffsetKey into ValueKey, starting at digit 'one to the right' = zeroth digit
3	OK^1 mod $OK^X = W^1$	OFFSET KEY ADD mode, OffsetKey plus ValueKey into OffsetKey, starting at digit 'one to the right' = zeroth digit

*NOTE: These cycle through as indicated in the default description above – these represent the first W created

Appendix A – QCy™ Stream Cipher

Qwyit Stream Cipher and Key Exchange/Update (QCy™)


Encrypt
Initial System Setup
Credentials issued OpenID, QK, EK
Initial Message Setup
Generate a new random salt (Open Return OR)
QCy™ Mission
The purpose of any encryption cipher should be the simplest, fastest, provably secure method allowing universal application everywhere. This QCy™ configuration delivers:
• World's Fastest
• World's Smallest, Most Efficient
• Provably Secure
• Quantum Safe, now and forever
QwyitChip™ QwyitSDK™
① Key Generation
Perform a MOD of each digit of OR, EK and QK, and a PDAF resulting in session start ValueKey (VK^C), OffsetKey (OK^C)
QK mod **OR** = VK^P
PDAF(EK, VK^P) = VK^C
EK mod **OR** = OK^P
PDAF(OK, OK^P) = OK^C
② Perform a MOD w/each digit of VK, OK, incrementing pointers Pc, Pv, Po
$VK^{Pc[1\ldots n]}{}_{Pv[1\ldots n]}$ Mod $OK_{Po[1\ldots n]} = W_{1\ldots n^2}$
③ Encrypt
Use W in XOR w/PT
$W_{1\ldots n^2} \oplus PT_{1\ldots n^2} = \mathbf{CT}_{1\ldots n^2}$
Or Decrypt using **CT**
④ Next Cycle
Perform PDAF Key Offset Add w/VK and OK
PDAF(OK^C, VK^P) = VK^{Next}
PDAF(VK^C, OK^P) = OK^{Next}
⑤ Send Message
Send **OR** and **CT** to the intended recipient using **OpenID**
Key Update
Secure periodic NO communication update of QK, EK
Decrypt

Send

- Generate random salt Open Return (OR) – a public value

1. QK mod **OR** = VK^P, PDAF(EK, VK^P) = VK^C and EK mod **OR** = OK^P, PDAF(QK, OK^P) = OK^C
2. SELECTION: $VK^{Pc[1\ldots n]}{}_{Pv[1\ldots n]}$ mod $OK_{Po[1\ldots n]} = W_{1\ldots n^2}$ Where pointer *Pv* and *Po* increment +1 through the entire key length for each cycle pointer *Pc.* If repeating, substitute VK_N and OK_N for each new cycle*
3. CIPHER: $W_{1\ldots n^2} \oplus PT_{1\ldots n^2} = \mathbf{CT}_{1\ldots n^2}$ repeating w/next cycle selection if more *PT*
4. UPDATE: PDAF(OK^C, VK^P) = VK^{Next} and PDAF(VK^C, OK^P) = OK^{Next}

- Send **OpenID**, **OR** and **CT** to Recipient

Receive

- Using the received random salt Open Return (OR)

1. QK mod **OR** = VK^P, PDAF(EK, VK^P) = VK^C and EK mod **OR** = OK^P, PDAF(QK, OK^P) = OK^C
2. SELECTION: $VK^{Pc[1\ldots n]}{}_{Pv[1\ldots n]}$ mod $OK_{Po[1\ldots n]} = W_{1\ldots n^2}$ Where pointer *Pv* and *Po* increment +1 through the entire key length for each cycle pointer *Pc.* If repeating, substitute VK_N and OK_N for each new cycle*
3. CIPHER: $W_{1\ldots n^2} \oplus PT_{1\ldots n^2} = \mathbf{CT}_{1\ldots n^2}$ repeating w/next cycle selection if more **CT**
4. UPDATE: PDAF(OK^C, VK^P) = VK^{Next} and PDAF(VK^C, OK^P) = OK^{Next}

[*NOTE: There are two (2) additional/recommended PDAF Selection calculations. See *QwyitCipher Ref Guide*]

Appendix B – Qwyit No Communication Key Update

There are two (2) ways to get new Master Keys for QCy™:

1. Static Update
    a. Using an out-of-band channel, the participants will share new 512-bit upper level, Master Keys at known intervals. This distribution is a public event, making the length of each 512-bit message chain known

2. Pseudo-Random Update
    a. Using any number of digits from any number of locations (existing *EK*, *QK*, Last *W*, etc.), and based on their values as stated by either the participant pair or the system owners, auto-update the Master Keys (*EK*, *QK*) using any combination of one-way Qwyit® primitives (PDAF/OWC, and/or Combine/Extract) – *without communication*

    For example:

    - Take the first 4 odd digits (1st,3rd,5th,7th) from the current *EK* and MOD16 add them together (1st + 3rd and 2nd + 4th).
        i. At this interval, both participants (without communication between them) update the keys using a *PDAF(EK, QK)* to twice the size of their total length (e.g., if the *EK* and *QK* are 256-bits each, perform the PDAF to 1024-bits.)
        ii. Then perform an OWC on this result, creating new *EK*, *QK* values
        iii. Use the new values for the next interval

    Other primitives may certainly be used and will maintain the underdetermined system; but the PDAF/OWC combination is 100% guaranteed, is an unsolvable one-way gate between key versions and is highly recommended.

    [NOTE: Using this Qwyit® key update method works for other secret key cryptosystems as well.]

The above updates can be performed forever without loss of entropy or key 'degradation' of any kind (certainly well beyond the update need in any real-world system) – See Appendix F.

[NOTE: It's also entirely simple and possible to pre-share an additional small set of random data (for example an extra 256-bits), that is used exclusively in either key update mode above to perform a PDAF with that small secret key and any-length of the existing key (and what to use can certainly be pseudo-randomly selected). This extra material would never be used in any *W* child/message key creations, extending the underdetermined system, and placing another one-way gate from any discovered messages or even stolen keys. This type of system flexibility is another example of the tremendous potential of Qwyit™.]

[NOTE: While the obvious difficulty/management of No Communication updates is staying in synch w/a shared key partner/device, it's quite straightforward to devise system 'recovery' strategies/options/methods. An example is keeping a set of the last *n*-updates, returning to a previous version upon discovery of mismatched messages. Each would need to be analyzed for 'weaknesses' (such as forcing an update by message theft/destruction, etc.), and the bottom line is that the security and advantage of No Communication makes it more than worthwhile to overcome any usage difficulty.]

Appendix C – Direct QCy™ Replacement of AES and Other Cipher Implementations

In order to aid proliferation of QCy™, the simplest implementation is to directly replace an existing one that uses AES-256. For these, QCy™ can be installed, and immediately begin operation. This is because most AES-256 implementations use one of the stream cipher modes of operation (CBC, CTR, OFB, etc.) – even the Authenticated Encryption modes do as well (GSM, etc.) And almost all of these are called with two parameters, an open IV and the key (both 256-bits).

These correspond directly to the QCy™ **OR** as the IV and the key can be duplicated as the *EK* and *QK*. Should the AES implementation allow easy update to include an additional key in the encrypt/decrypt call, then that is preferred and these would be *EK* and *QK*. It's possible in the single key operation to modify the single key creating an 'original' 2nd key (using PDAF/OWC, etc.), but the security isn't improved so this is only of value should the implementation not be publicly available (this is the case in most IoT implementations, where one has to perform serious detailed investigation to understand the Bluetooth security setup).

For other existing cipher implementations the same input mapping should be performed (**OR** as any public seed or initialization vector, and secret keys as *EK*, *QK*). It should be remembered that QCy™ doesn't have any length requirements, so other cipher mapping using different key lengths can still be accomplished.

Appendix D – QCy™ PDAF_SEC Security Discussion

As stated in the Introduction, Qwyit® is a Security Engineering company. We're not cryptographers, we're Information Theorists. Therefore, as Gilbert Vernam before us, we're going to leave it to a 'security industry' mathematician like Shannon, to write up any QCy™ 'proof of security'. We're going to concentrate on the engineering and delivery of our version of Shannon's *Perfect Secrecy Ideal System* because the world should benefit from its superior properties. The following is a general outline and discussion of QCy™ security.

We base all of our accomplishments in true engineering spirit and fashion. We build security that works, because we:

1. Started from the beginning – What, *exactly*, is 'privacy' and 'security' in an all-digital, always-on, everything connected world?
2. Learned 'How it actually works now' – What does a network really do; the actual science of how it operates, *and the purpose of it in relation to the people who use them* <u>*today*</u>
3. Distilled 'What security and privacy *should* be' – If you take any one of those real networks, and the operators, the users, the criminals, putting all of their activities and requirements into the Same Big, Universal Example, *what would actually deliver the proper, required security and privacy?*
4. We tested that network, then additional networks, then designed, then retested, then redesigned, then retested again and again…reconfigured it over and over until *we found a way to deliver a real world product that meets those real world requirements*
5. That product is an Authentication and Data Security protocol called Qwyit™, along with the accompanying QCy™ cipher – and they have the *Real World* required properties: speed, efficiency, flexibility, provable security. They deliver universally, and deserve worldwide proliferation

We can, and have, built it where the *speed* has been benchmarked and can be verified (World's Fastest). We have, and it can be verified, demonstrated our build's *efficiency* in bandwidth, code space, universal application (World's Most Efficient). We have, and it can be verified, displayed our flexibility in our portfolio of products within almost every major network/market application (*QwyitChip*™, *QwyitSDK*™, *QwyitKey*™*, QwyitTalk*™, *QwyitStore*™, *QwyitCard*™*, QwyitCash*™).

It's that last feature/benefit/property: provable security. Cryptographers can/will instantly – incorrectly, as, and will be shown – dismiss short key OTPs as flawed from Shannon. Business leadership will defer to cryptographers. End Users defer to Biz leaders; investor's defer to them all. And all of them have forgotten that Shannon's short key *Ideal System* <u>does exist</u> – he's shown it, detailed it:

First, here's *Perfect Secrecy*

""Perfect Secrecy" is defined by requiring of a system that after a cryptogram is intercepted by the enemy the *a posteriori* probabilities of this cryptogram representing various messages be identically the same as the *a priori* probabilities of the same messages before the interception. It is shown that perfect secrecy is possible but requires, if the number of messages is finite, the same number of possible keys. If the message is thought of as being constantly generated at a given "rate" R (to be defined later), key must be generated at the same or a greater rate."

This is exactly what the PDAF does: generates the same number of key bits for every plaintext bit. The question is whether starting from a 'finite key', this generation is 'Ideal'. Here's the *Ideal System*:

"It is possible to construct secrecy systems with a finite key for certain "languages" in which the equivocation does not approach zero as $N \to \infty$. In this case, no matter how much material is intercepted, the enemy still does not obtain a unique solution to the cipher but is left with many alternatives, all of reasonable probability. Such systems we call *ideal* systems. It is possible in any language to approximate such behavior—i.e., to make the approach to zero of $H(N)$ recede out to arbitrarily large N."

Shannon goes on to define/explain the structure of an *Ideal System*:

"To approximate the ideal equivocation, one may first operate on the message with a transducer which removes all redundancies. After this almost any simple ciphering system—substitution, transposition, Vigen`ere, etc., is satisfactory. The more elaborate the transducer and the nearer the output is to the desired form, the more closely will the secrecy system approximate the ideal characteristic."

This is exactly our security engineering accomplishment: Our PDAF, used in every step of the QCy™ crypto*system* that generates the endless *Perfect Secrecy* keys, is Shannon's "*transducer*" that *delivers, not 'approximates', "the ideal characteristic"*. [Our *transducer* operates on the key not the message; which led to our breakthroughs.] As part of that accomplishment, it isn't "*elaborate*" at all – it's incredibly simple and straightforward – which realizes QCy™'s most important goal: Real World speed and efficiency. The QCy™ cipher is then the same "*simple ciphering system*" used in *Perfect Secrecy* – a simple plaintext XOR with the endless key.

What we've accomplished is a finite key *Ideal System* for any bit "*language"* where the PDAF generates *Perfect Secrecy* endless keys: QCy™ is underdetermined at every step (PDAF at the Start, in key Selection, and next key Update), producing the required property result: there are multiple possible answers throughout the entire use of the crypto*system*.

Theorem 1. *The PDAF delivers multiple possible solutions throughout the entire QCy™ cryptosystem, realizing Shannon's stated, proved and exampled Perfect Secrecy in an Ideal System. Therefore QCy™ is Provably Secure*.

While Shannon stated, proved and exampled a *Perfect Secrecy Ideal System*, he concluded that other than working with "*natural languages*":

"The complexity of the system needed usually goes up rapidly when we attempt to do this, however. It is not always possible to attain actually the ideal characteristic with any system of finite complexity"

In building our *Perfect Secrecy Ideal QCy™ System*, we solved Shannon's "*complexity*" issues for any bit stream '*language'* by engineering two distinct cryptographic innovations:

1. *Random Rearrangement (RR)* - digit position manipulation, based upon random inputs and modular arithmetic properties, including the use of *remaining unused digit position values within a given input*, can be performed infinitely without any resulting randomness degradation; e.g., Keys created using *RR* from existing random keys are unique, random and possess all the characteristics of independently generated random keys (See Appendix F for examples)

2. As shown in Appendix B, and reliant on *RR*, updating participant keys using a PDAF/OWC combination can be performed synchronously, without communication, and the security is perfectly one-way: past keys cannot be *exclusively determined* (Qwyit creates an underdetermined equation set that produces a large keyspace of incorrect values, a small set of valid values and one correct value.) Any system level keys – Master Authentication Keys, Session Start Keys, etc. – can forever be updated in a one-way, perfect, fast method *without any required communication between key partners*

When these two techniques are used as the fundamental building blocks of our Qwyit Protocol and QCy™ cipher, our crypto*system* presents the identical properties of *Perfect Secrecy*; and therefore a finite key *Ideal System*. There is never total discernment of any result.

The PDAF *RR* can continually present a one-way, random stream of underdetermined keys that *will never produce only one possibility.* The QCy™ cipher uses the PDAF three times for Start, Selection and Update – and none of these results are ever *exclusively determinable*. Yes, one can search the entire finite key space and return the correct result. But one is never certain which result out of the valid set is the correct one, as the PDAF will *always return an underdetermined one-way small set of valid possibilities*. Which continues during a cycle, into the next cycle, the next update, the next OR-seeded session – and *since it's possible that any session has been Master Key updated with no communication such that even an all-powerful adversary does not know this has occurred, QCy will never produce a known broken singular result*.

Theorem 2. *The QCy™ crypto<u>system</u>, being Provably Secure, produces the first and only Perfect Cross Security*

Let's review QCy™ for Perfect Security Cross attacks:

- UP – Continuously defeated by the underdetermined, irreversible PDAF Offset Key Add mode
- RIGHT – Even with previous message knowledge, including *all* values except the Master Keys, new message breaks are defeated by the same PDAF/OR reseed
- LEFT – Same as RIGHT
- DOWN – With only some PT knowledge and corresponding *W* section, the *VK/OK* keys all remain underdetermined for other sections, and maintain message key integrity

See Appendix F for empirical demonstrations of *Perfect Secrecy* results, showing that all four Cross direction attacks are defeated. An example of the system's combinatorial probabilities is included.

In summary, this is the best cryptographic engineering can accomplish: the real world implementation of Shannon's *Perfect Secrecy Ideal System*. At the end of this section, and years of testing, including independent cryptographic reviews and NIST Lightweight Cryptography accepted submission (2015 – see Qwyit.com), *one thing is undeniably true*:

QCy™ is faster, more efficient, more flexible and indicatively *more* secure than any current system.

Appendix E – QCy™ Stream Cipher Reference Code

The following two reference code Qwyit™ functions are provided: PDAF and PDAF_SEC. The PDAF_SEC is the Qwyit encryption/decryption function. The PDAF is used for key updates:

FUNCTION: Position Digit Algebra Function - (PDAF)

```
'
' NAME:     PDAF
'
' PURPOSE:  Returns a key expansion based on single or dual key input
'
' TYPE:     Qwyit specific function - uses strings/byte return
'
' CALL:     PDAF_SEC(sValueKey, nMode, Optional nDigits, Optional sOffsetKey)
'           where sValueKey is the Value Key from which to select,
'           nMode is whether to perform a ValueKey Offset add (nMode = 1) or
'              perform the default dual-key pointer add (nMode = 0)
'           Optional nDigits is how many 4-bit digits to return (If 0, then return all cycles - which is the
'                     length of the ValueKey squared)
'           Optional sOffsetKey is the key which points into the ValueKey for selection (if entered, it
'                     MUST be the same size as the ValueKey)
'                     (if not entered, the OffsetKey will be the ValueKey)
'
' RTRN:     Byte result of the key material - either all cycles [len(ValueKey] squared), or nDigits
'
' ERROR:    Null return
'
' Example:  PDAF("0123456789ABCDEF", 0, 0, "FEDCBA9876543210") returns
"FFFFFFFFFFFFFFFF0000000000000000111111111111111122222222222222223333333333333333444444444444444455555555555555556666666666666666
677777777777777778888888888888888999999999999999AAAAAAAAAAAAAAAABBBBBBBBBBBBBBBBCCCCCCCCCCCCCCCCDDDDDDDDDDDDDDDD
EEEEEEEEEEEEEEEE"
' Example:  PDAF("FB382C001A", 0, 30, "CC69100AB4") returns "B7913C0ACE7FEBD00B53F4851014AF"
' Example:  PDAF("0123456789ABCDEF", 1, 0, "FEDCBA9876543210") returns
"0123456789ABCDEF23456789ABCDEF01456789ABCDEF01236789ABCDEF01234589ABCDEF01234567ABCDEF0123456789CDEF0123456789ABEF01234
56789ABCD0123456789ABCDEF23456789ABCDEF01456789ABCDEF01236789ABCDEF01234589ABCDEF01234567ABCDEF0123456789CDEF0123456789
ABEF0123456789ABCD"
' Example:  PDAF("FB382C001A", 1, 30, "CC69100AB4") returns "7DD02C010CDF74C01B5BF8D811B92B"
'
'
' Test Vector: The examples are the test vectors
```

```
'
Public Function PDAF(sValueKey As String, nMode As Integer, Optional nDigits As Integer, Optional sOffsetKey As String) As Variant

Dim t_Key() As Byte
Dim nKey As Long
Dim t_sKey As String
Dim sKey As String
Dim sAddKey As String
Dim sPointKey As String
Dim sHoldTemp As String
Dim sHoldFirst As String
Dim sTmpVal As String

If sValueKey = Null Or sValueKey = "" Then
   PDAF = ""
   Exit Function
Else
   If sOffsetKey = "" Or sOffsetKey = Null Then
      sOffsetKey = sValueKey
   End If
End If

sPointKey = sOffsetKey
sAddKey = sValueKey
nLen = Len(sAddKey)

If nDigits = 0 Then
   nN_Max = nLen * nLen
Else
   nN_Max = nDigits
End If
ReDim t_Key(nN_Max)

If nMode = 0 Then   'Dual Key Add mode
   p = 1 'Pointer starts at 1
   c = 0 'Cycle starts at 0
   nN = 0  'Counter for how much key material
   Do
     Formula1 = Val("&H" & Mid(sPointKey, p, 1))
     Formula2 = Val("&H" & Mid(sAddKey, p + c, 1))

     t_Key(nN) = Asc(Hex((Formula1 + Formula2) Mod 16))
```

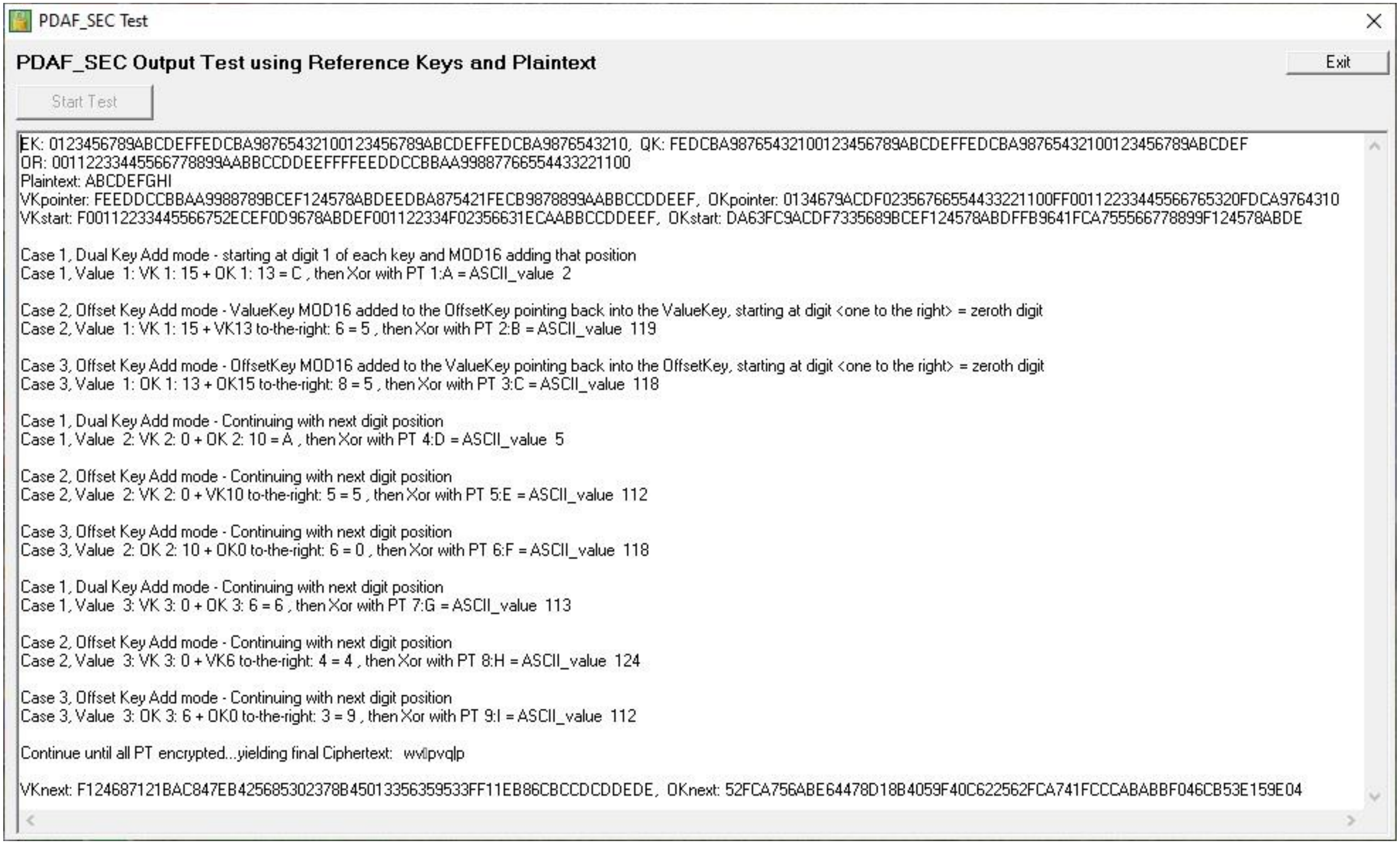
```
        nN = nN + 1
        If nN = nN_Max Then
            Exit Do
        End If

        p = p + 1

        If p > nLen Then
            p = 1
            c = c + 1
        End If
    Loop
Else    'nMode <>0 so do the Key Offset Add mode
    'Expand the keys for ease in going 'round the corner' in digit selection
    While Len(sAddKey) < (2 * nLen) + 17
        sAddKey = Trim(Trim(sAddKey) & Trim(sAddKey))
    Wend
    nLen2 = Len(sPointKey)
    While Len(sPointKey) < (2 * nLen2) + 17
        sPointKey = Trim(Trim(sPointKey) & Trim(sPointKey))
    Wend

    p = 1 'Pointer starts at 1
    c = 0 'Cycle starts at 0
    nN = 0  'Counter for how much key material
    Do
       Formula1 = Val("&H" & Mid(sAddKey, p + c, 1))
       'nOffset = Val("&H" & Mid(sPointKey, p, 1)) + 1
       'Formula2 = Val("&H" & Mid(sAddKey, p + nOffset + c, 1))
       Formula2 = Val("&H" & Mid(sAddKey, p + Val("&H" & Mid(sPointKey, p, 1)) + 1 + c, 1))

       t_Key(nN) = Asc(Hex((Formula1 + Formula2) Mod 16))

        nN = nN + 1
        If nN = nN_Max Then
            Exit Do
        End If

        p = p + 1

        If p > nLen Then
            p = 1
            c = c + 1
        End If
    Loop
End If
'Return a byte
PDAF = t_Key()

End Function
```

FUNCTION: Position Digit Algebra Function Security - Enc/Dec (PDAF_SEC)

```
'
' NAME:     PDAF_SEC
'
' PURPOSE:  Returns an XOR result of the Plaintext w/the PDAF never-ending result
'
' TYPE:     Qwyit specific function - uses string inputs, byte output
'
' CALL:     PDAF_SEC(sValueKey, , sKeyResult, sTargettext, sOR, [sOffsetKey])
'           where sValueKey is the Value Key from which to select [This is the Qwyit Auth Key, EK],
'           sTargettext is either the plaintext to be encrypted or the ciphertext to be decrypted,
'           sOR is the OpenReturn (an initialization vector for cycle updates)
'           Optionally, sOffsetKey is the key from which to set the offset (If no value, sValueKey will be duplicated for use as the OffsetKey - if included, it MUST be the same length as the ValueKey)
'               [This is the Qwyit Auth Key, QK]
'
' RTRN:     Byte result of the key material
'
' ERROR:    Null return
'
' Example:  The test vector file is the example
'
' Test Vector: The test vector file is the test vector
'
Public Function PDAF_SEC(sValueKey As String, sTargetText As String, sOR As String, Optional sOffsetKey As String) As Variant

Dim t_Key() As Byte
Dim PDAF_AddKey() As Byte
Dim PDAF_PointKey() As Byte
Dim sAddKey As String
Dim sPointKey As String
Dim PDAF_AddKeyPointer() As Byte
Dim PDAF_PointKeyPointer() As Byte
Dim sValueKeyPointer As String
Dim sOffsetKeyPointer As String
Dim nCases As Integer

If sValueKey = Null Or sValueKey = "" Or sOR = "" Or sOR = Null Then
   PDAF_SEC = ""
   Exit Function
Else
   If sOffsetKey = "" Or sOffsetKey = Null Then
      sOffsetKey = sValueKey
   End If
End If

'SESSION START: Control step for reconfiguring the start keys
sValueKeyPointer = MOD16(sOffsetKey, sOR)   'The OffsetKey is QK
sOffsetKeyPointer = MOD16(sValueKey, sOR)   'The ValueKey is EK

PDAF_AddKeyPointer = PDAF(sValueKey, 1, 64, sValueKeyPointer)
PDAF_PointKeyPointer = PDAF(sOffsetKey, 1, 64, sOffsetKeyPointer)
   'This is just to get back to strings...but shouldn't be needed in other platforms...(slows things down! :(
   For lCount = 0 To UBound(PDAF_PointKeyPointer) - 1
      sAddKey = sAddKey & Chr(PDAF_AddKeyPointer(lCount))
      sPointKey = sPointKey & Chr(PDAF_PointKeyPointer(lCount))
   Next

'Set the length of bytes returned
nN_Max = Len(sTargetText)
ReDim t_Key(nN_Max)

nLen = Len(sAddKey)
While Len(sAddKey) < (2 * nLen) + 17
   sAddKey = Trim(Trim(sAddKey) & Trim(sAddKey))
Wend
nLen2 = Len(sPointKey)
While Len(sPointKey) < (2 * nLen2) + 17
   sPointKey = Trim(Trim(sPointKey) & Trim(sPointKey))
Wend

p = 1 'Pointer starts at 1
C = 0 'Cycle starts at 0
nN = 0  'Counter for how much key material

Do
      'Beginning of CASES
      'CASE #1 - DUAL KEY ADD mode, ValueKey plus OffsetKey, starting at digit 1 of each and adding that position
      'VK1 + OK1 = W1
      '   Using the AddKey for cycling through
         'SELECTION (Just for definition...it is performed all in one Select & Cipher command next...)
         'Formula1 = Val("&H" & Mid(sPointKey, p, 1))
         'Formula2 = Val("&H" & Mid(sAddKey, p + C, 1))
         'sTmpVal = Hex((Formula1 + Formula2) Mod 16)

         'CIPHER (encrypt/decrypt)
```

```
        'IF selection performed above...
            't_Key(nN) = Asc(sTmpVal) Xor Asc(Mid(sTargetText, nN+1, 1))
        'For speed, perform SELECTION and CIPHER in one command...[not really much faster...]
        t_Key(nN) = Asc(Hex((Val("&H" & Mid(sPointKey, p, 1)) + Val("&H" & Mid(sAddKey, p + C, 1))) Mod 16)) Xor Asc(Mid(sTargetText, nN + 1, 1))
        nN = nN + 1
        If nN = nN_Max Then
            'We're done w/all the TargetText
            Exit Do
        End If
    'END Of CASE #1

    'CASE #2 - OFFSET KEY ADD mode, ValueKey plus OffsetKey into ValueKey, starting at digit 'one to the right' = zeroth digit
    'VK1 + VKx = W2
    '   Using the AddKey for cycling through
        'SELECTION (Just for definition...it is performed all in one Select & Cipher command next...)
        'Formula1 = Val("&H" & Mid(sAddKey, p + Val("&H" & Mid(sPointKey, p, 1)) + 1, 1))
        'Formula2 = Val("&H" & Mid(sAddKey, p + C, 1))
        'sTmpVal = Hex((Formula1 + Formula2) Mod 16)

        'CIPHER (encrypt/decrypt)
        'IF selection performed above...
            't_Key(nN) = Asc(sTmpVal) Xor Asc(Mid(sTargetText, nN+1, 1))
        'For speed, perform SELECTION and CIPHER in one command...
        t_Key(nN) = Asc(Hex((Val("&H" & Mid(sAddKey, p + Val("&H" & Mid(sPointKey, p, 1)) + 1, 1)) + Val("&H" & Mid(sAddKey, p + C, 1))) Mod 16)) Xor
Asc(Mid(sTargetText, nN + 1, 1))
        nN = nN + 1
        If nN = nN_Max Then
            'We're done w/all the TargetText
            Exit Do
        End If
    'END Of CASE #2

    'CASE #3 - OFFSET KEY ADD mode, OffsetKey plus ValueKey into OffsetKey, starting at digit 'one to the right' = zeroth digit
    'OK1 + OKx = W3
    '   Using the PointKey for cycling through
        'SELECTION (Just for definition...it is performed all in one Select & Cipher command next...)
        'Formula1 = Val("&H" & Mid(sPointKey, p + Val("&H" & Mid(sAddKey, p, 1)) + 1, 1))
        'Formula2 = Val("&H" & Mid(sPointKey, p + C, 1))
        'sTmpVal = Hex((Formula1 + Formula2) Mod 16)

        'CIPHER (encrypt/decrypt)
        'IF selection performed above...
            't_Key(nN) = Asc(sTmpVal) Xor Asc(Mid(sTargetText, nN+1, 1))
        'For speed, perform SELECTION and CIPHER in one command...
        t_Key(nN) = Asc(Hex((Val("&H" & Mid(sPointKey, p + Val("&H" & Mid(sAddKey, p, 1)) + 1, 1)) + Val("&H" & Mid(sPointKey, p + C, 1))) Mod 16)) Xor
Asc(Mid(sTargetText, nN + 1, 1))
        nN = nN + 1
        If nN = nN_Max Then
            'We're done w/all the TargetText
            Exit Do
        End If
    'END Of CASE #3
  p = p + 1
  'UPDATE test
  If p > nLen Then
    p = 1
    C = C + 1
    If C = nLen Then
        'Ok, we're at the end of the cycles, and since still need more results for enc/dec, reseed keys
        C = 0
        'PERFORM UPDATE
        PDAF_PointKey = PDAF(sAddKey, 1, 64, sOffsetKeyPointer)
        PDAF_AddKey = PDAF(sPointKey, 1, 64, sValueKeyPointer)
        sAddKey = ""
        sPointKey = ""
        'This is just to get back to strings...but shouldn't be needed in other platforms...(slows things down! :(
        For lCount = 0 To UBound(PDAF_PointKey) - 1
            sAddKey = sAddKey & Chr(PDAF_PointKey(lCount))
            sPointKey = sPointKey & Chr(PDAF_AddKey(lCount))
        Next

        'Then stretch out the keys for easy 'around the corner' calculations
        While Len(sAddKey) < (2 * nLen) + 17
            sAddKey = Trim(Trim(sAddKey) & Trim(sAddKey))
        Wend
        While Len(sPointKey) < (2 * nLen2) + 17
            sPointKey = Trim(Trim(sPointKey) & Trim(sPointKey))
        Wend
    End If
  End If
Loop
'Return a byte
PDAF_SEC = t_Key()
End Function
```

The picture below shows the reference PDAF_SEC function operation (all three Selection Cycles, XOR cipher and Key Update results) from an example key set and plaintext (*VKstart* and *OKstart* created as per Session Start; MOD16 w/**OR**, then PDAF mixed w/*EK,QK*.)



PDAF_SEC Test

PDAF_SEC Output Test using Reference Keys and Plaintext

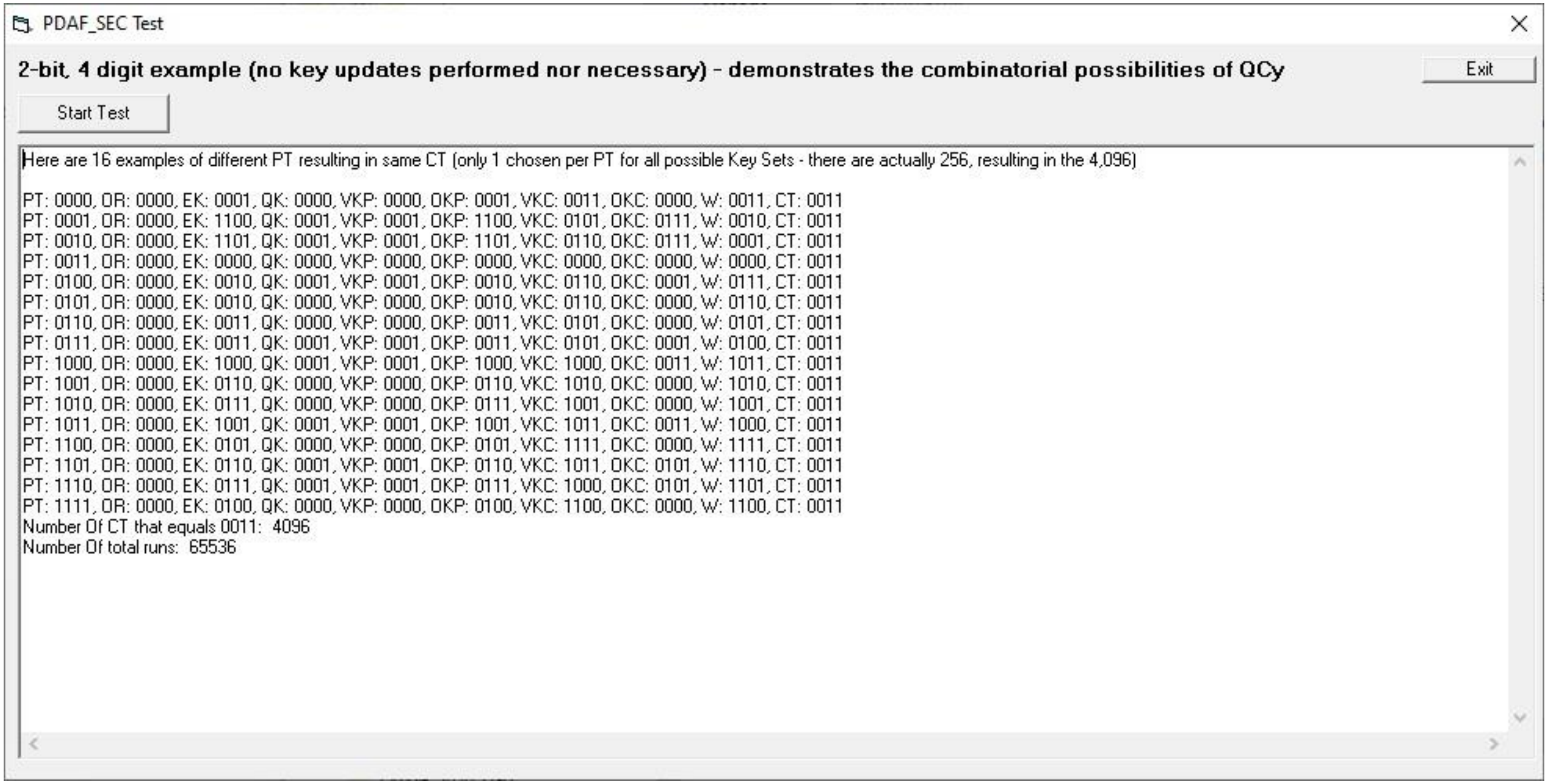
Exit

Start Test

```
EK: 0123456789ABCDEFFEDCBA98765432100123456789ABCDEFFEDCBA9876543210,  QK: FEDCBA98765432100123456789ABCDEFFEDCBA98765432100123456789ABCDEF
OR: 00112233445566778899AABBCCDDEEFFFFEEDDCCBBAA99887766554433221100
Plaintext: ABCDEFGHI
VKpointer: FEEDDCCBBAA9988789BCEF124578ABDEEDBA875421FECB9878899AABBCCDDEEF,  OKpointer: 0134679ACDF0235676655443322110OFF001122334455667653 20FDCA9764310
VKstart: F00112233445566752ECEF0D9678ABDEF001122334F02356631ECAABBCCDDEEF,  OKstart: DA63FC9ACDF7335689BCEF124578ABDFFB9641FCA755566778899F124578ABDE

Case 1, Dual Key Add mode - starting at digit 1 of each key and MOD16 adding that position
Case 1, Value  1: VK 1: 15 + OK 1: 13 = C , then Xor with PT 1:A = ASCII_value  2

Case 2, Offset Key Add mode - ValueKey MOD16 added to the OffsetKey pointing back into the ValueKey, starting at digit <one to the right> = zeroth digit
Case 2, Value  1: VK 1: 15 + VK13 to-the-right: 6 = 5 , then Xor with PT 2:B = ASCII_value  119

Case 3, Offset Key Add mode - OffsetKey MOD16 added to the ValueKey pointing back into the OffsetKey, starting at digit <one to the right> = zeroth digit
Case 3, Value  1: OK 1: 13 + OK15 to-the-right: 8 = 5 , then Xor with PT 3:C = ASCII_value  118

Case 1, Dual Key Add mode - Continuing with next digit position
Case 1, Value  2: VK 2: 0 + OK 2: 10 = A , then Xor with PT 4:D = ASCII_value  5

Case 2, Offset Key Add mode - Continuing with next digit position
Case 2, Value  2: VK 2: 0 + VK10 to-the-right: 5 = 5 , then Xor with PT 5:E = ASCII_value  112

Case 3, Offset Key Add mode - Continuing with next digit position
Case 3, Value  2: OK 2: 10 + OK0 to-the-right: 6 = 0 , then Xor with PT 6:F = ASCII_value  118

Case 1, Dual Key Add mode - Continuing with next digit position
Case 1, Value  3: VK 3: 0 + OK 3: 6 = 6 , then Xor with PT 7:G = ASCII_value  113

Case 2, Offset Key Add mode - Continuing with next digit position
Case 2, Value  3: VK 3: 0 + VK6 to-the-right: 4 = 4 , then Xor with PT 8:H = ASCII_value  124

Case 3, Offset Key Add mode - Continuing with next digit position
Case 3, Value  3: OK 3: 6 + OK0 to-the-right: 3 = 9 , then Xor with PT 9:I = ASCII_value  112

Continue until all PT encrypted...yielding final Ciphertext:   wv▯pvqlp

VKnext: F124687121BAC847EB425685302378B45013356359533FF11EB86CBCCDCDDEDE,  OKnext: 52FCA756ABE64478D18B4059F40C622562FCA741FCCCABABBF046CB53E159E04
```

"To prove that a scheme is perfectly secure, you must be able to show that for any pair of messages, the probability that they map to a given ciphertext is identical. This would usually be a straightforward probability argument. One example of a perfectly secure system is a one-time pad." – CryptoProofs

As engineers, we firmly believe in real world testing and examples. Scale model testing is an excellent method for distilling methodology down to the core capability as well as highlighting problems. In performing a 2-bit, 4-digit version of QCy™, the provable security is apparent and obvious. Here's a screen capture of a program written to execute this scaled down version:



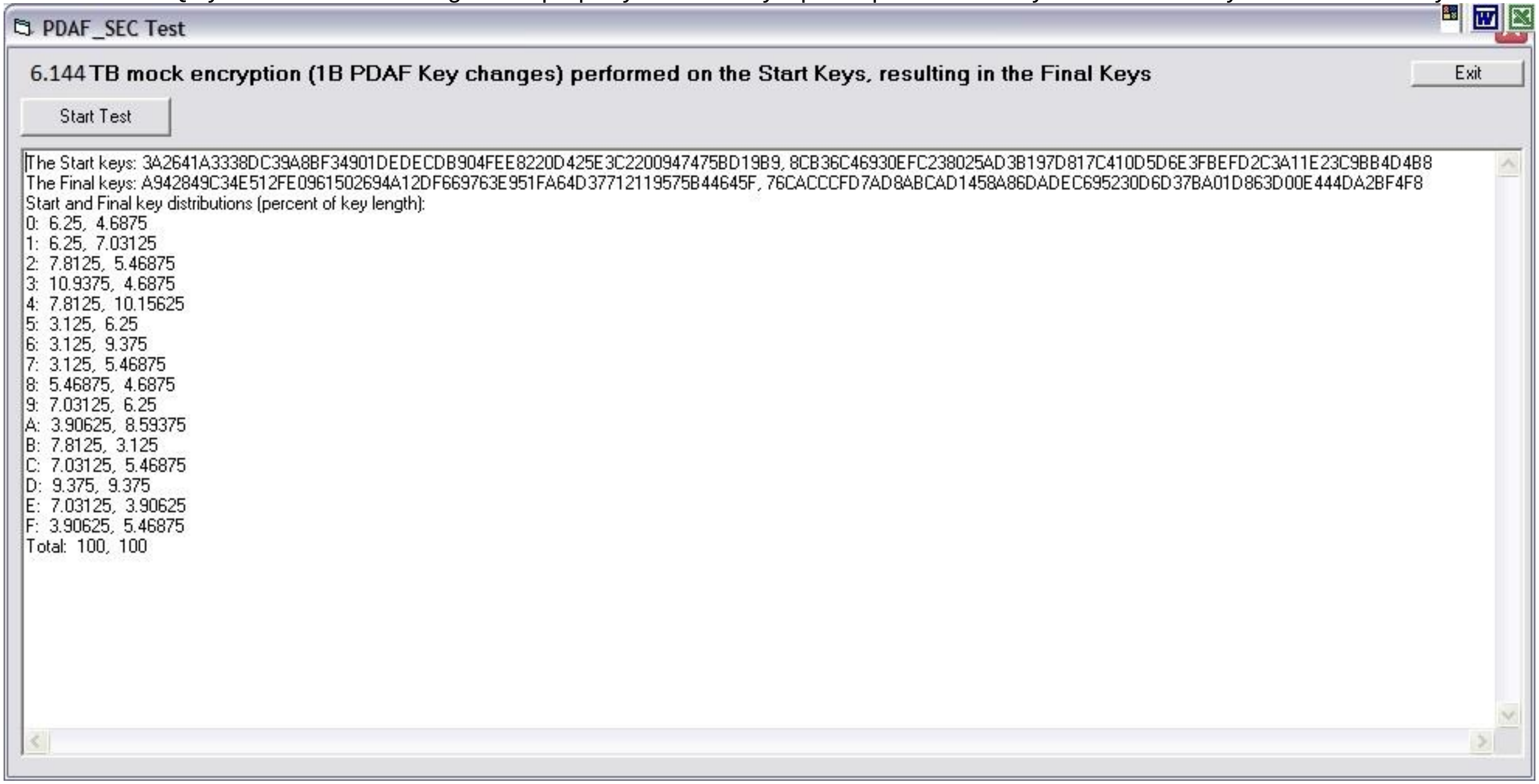
2-bit, 4 digit example (no key updates performed nor necessary) - demonstrates the combinatorial possibilities of QCy

```
Here are 16 examples of different PT resulting in same CT (only 1 chosen per PT for all possible Key Sets - there are actually 256, resulting in the 4,096)

PT: 0000, OR: 0000, EK: 0001, QK: 0000, VKP: 0000, OKP: 0001, VKC: 0011, OKC: 0000, W: 0011, CT: 0011
PT: 0001, OR: 0000, EK: 1100, QK: 0001, VKP: 0001, OKP: 1100, VKC: 0101, OKC: 0111, W: 0010, CT: 0011
PT: 0010, OR: 0000, EK: 1101, QK: 0001, VKP: 0001, OKP: 1101, VKC: 0110, OKC: 0111, W: 0001, CT: 0011
PT: 0011, OR: 0000, EK: 0000, QK: 0000, VKP: 0000, OKP: 0000, VKC: 0000, OKC: 0000, W: 0000, CT: 0011
PT: 0100, OR: 0000, EK: 0010, QK: 0001, VKP: 0001, OKP: 0010, VKC: 0110, OKC: 0001, W: 0111, CT: 0011
PT: 0101, OR: 0000, EK: 0010, QK: 0000, VKP: 0000, OKP: 0010, VKC: 0110, OKC: 0000, W: 0110, CT: 0011
PT: 0110, OR: 0000, EK: 0011, QK: 0000, VKP: 0000, OKP: 0011, VKC: 0101, OKC: 0000, W: 0101, CT: 0011
PT: 0111, OR: 0000, EK: 0011, QK: 0001, VKP: 0001, OKP: 0011, VKC: 0101, OKC: 0001, W: 0100, CT: 0011
PT: 1000, OR: 0000, EK: 1000, QK: 0001, VKP: 0001, OKP: 1000, VKC: 1000, OKC: 0011, W: 1011, CT: 0011
PT: 1001, OR: 0000, EK: 0110, QK: 0000, VKP: 0000, OKP: 0110, VKC: 1010, OKC: 0000, W: 1010, CT: 0011
PT: 1010, OR: 0000, EK: 0111, QK: 0000, VKP: 0000, OKP: 0111, VKC: 1001, OKC: 0000, W: 1001, CT: 0011
PT: 1011, OR: 0000, EK: 1001, QK: 0001, VKP: 0001, OKP: 1001, VKC: 1011, OKC: 0011, W: 1000, CT: 0011
PT: 1100, OR: 0000, EK: 0101, QK: 0000, VKP: 0000, OKP: 0101, VKC: 1111, OKC: 0000, W: 1111, CT: 0011
PT: 1101, OR: 0000, EK: 0110, QK: 0001, VKP: 0001, OKP: 0110, VKC: 1011, OKC: 0101, W: 1110, CT: 0011
PT: 1110, OR: 0000, EK: 0111, QK: 0001, VKP: 0001, OKP: 0111, VKC: 1000, OKC: 0101, W: 1101, CT: 0011
PT: 1111, OR: 0000, EK: 0100, QK: 0000, VKP: 0000, OKP: 0100, VKC: 1100, OKC: 0000, W: 1100, CT: 0011
Number Of CT that equals 0011:  4096
Number Of total runs:  65536
```

This small version execution shows that for every possible PT (there are 16; 0000, 0001, 0010, etc.), calculated with every possible OR (16), using every possible QK (16) and EK (16), there are 4,096 identical PT-to-CT mappings out of the 65,536 total possibilities.

The program shows one for every same PT, where there are 256; e.g., for every different PT against all possible key sets (OR, QK, EK, which is 16^3 or 4,096), there are 256 identical PT-to-CT mappings. Since there are 16 different possible PTs, there are 16*256 identical sets (4,096) out of the total 65,536 different PT/OR/QK/EK pairings. This empirical QCy™ execution demonstrates the combinatorial probabilities at any key size. The larger the key, the smaller the percentage valid set of possible answers (only one is ever correct). For any particular ciphertext, there is a straightforward probability mapping of multiple plaintexts: QCy™ is provably secure and delivers a *true Perfect Secrecy Ideal System.*

This is demonstrably true for any QCy™ Session Start, Selection Case and Cipher. The only remaining security discussion is whether our innovative QCy™ *Random Rearrangement* property used in Key Update provides a truly new random key set for the next cycle.



6.144 TB mock encryption (1B PDAF Key changes) performed on the Start Keys, resulting in the Final Keys

```
The Start keys: 3A2641A3338DC39A8BF34901DEDECDB904FEE8220D425E3C2200947475BD19B9, 8CB36C46930EFC238025AD3B197D817C410D5D6E3FBEFD2C3A11E23C9BB4D4B8
The Final keys: A942849C34E512FE0961502694A12DF669763E951FA64D37712119575B44645F, 76CACCCFD7AD8ABCAD1458A86DADEC695230D6D37BA01D863D00E444DA2BF4F8
Start and Final key distributions (percent of key length):
0:  6.25,  4.6875
1:  6.25,  7.03125
2:  7.8125,  5.46875
3:  10.9375,  4.6875
4:  7.8125,  10.15625
5:  3.125,  6.25
6:  3.125,  9.375
7:  3.125,  5.46875
8:  5.46875,  4.6875
9:  7.03125,  6.25
A:  3.90625,  8.59375
B:  7.8125,  3.125
C:  7.03125,  5.46875
D:  9.375,  9.375
E:  7.03125,  3.90625
F:  3.90625,  5.46875
Total:  100,  100
```

The above is a screen capture of a small program written to perform 1Billion PDAF *Random Rearrangements (RR)* using the PDAF function as called for in the PDAF_SEC QCy™ encryption key Update. As shown, if the keys are random to start, this PRNG process ends with random keys; the 1B updates represent 6.144TB of encrypted data (using all 3 Selection Cases). Key updates in this manner provide a provably mathematic one-way gate; and have long-lasting system capability, limiting the number of times a Static new key delivery is 'required'. All of the PDAF key changes pass statistical Random tests (NIST Statistical Test Suite tested and verified).

Conclusion

It's obvious from these examples, the underdetermined every-step PDAF processing, and even all-powerful *N*→∞ capture of *every* message, that QCy™ is, indeed, an *Ideal System*. An attacker will never positively gain knowledge through the system from any brute force result that *it is correct*; the work expands to ∞ with every possible unknown PDAF/OWC Master Key update having to be re-started from the beginning and computed through to the present, only to be left w/still multiple possible results.

The point is proven: Identical mapping of two different PTs to a single CT means *Perfect Secrecy*. QCy™ is an *Ideal System*.

And although this is an incredibly powerful – and final – cryptographic result, the *main supremacy of QCy™ is Real World <u>practicality</u>:*

QCy™ is the fastest, most efficient, most secure encryption engine – *<u>Use It!</u>*

Appendix G – References, Notes

References

Shannon, C. (1945). *Communication Theory of Secrecy Systems*

McGough, P. (2010). *Qwyit Protocol Reference* (Latest edition, June 2020, V3.0)

Notes

NOTE1: Cryptography has misapplied the term '*Authenticated Encryption'* such that Qwyit® is forced to 'invent' a new term *Authentic Encryption* – which is exactly what one thinks 'Authentic*ated* Encryption' means (but doesn't): Encryption that is Authentic, from the source who owns the encryption keys; e.g., the digital communication method of Authentication has been combined with the method of hiding the data (encryption). We will now call this *'Authentic Encryption'.*

'Authentic*ated* Encryption' is misnamed by cryptographers to mean encryption that has not been corrupted, e.g., has been received with proper message *integrity*, which is what *authenticity* means (not *authentication*). They should have called that either 'Authenticity Encryption' or 'Integrity Encryption' – not 'Authentic*ated*'. But now we have a method that actually meets what people think – and need – when they use *Authentic Encryption.*

NOTE2: Perfect Secrecy is 'susceptible' to a Known Plaintext 'attack'. The reason those are in quotes, is because Perfect Secrecy never reveals the contents – *but,* it may be used in situations where there is positively known plaintext (greetings, etc.) When this is the case, the message key used for that will be revealed (*not* broken). Since in true Perfect Secrecy all other message bits are unique and unbreakable, it is incumbent on our finite-key method (*Ideal System*) to remain Perfectly Secret, never leading to breaking/revealing any unknown remaining parts of the message as well as the other three arms of the cross.