T-Proof

Secure Communication via Non-Algorithmic Randomization

Proving Possession of Data to a Party in possession of same data

Gideon Samid
Department of Electrical Engineering and Computer Science
Case Western Reserve University
BitMint, LLC
Gideon@BitMint.com

Abstract: shared random strings are either communicated or recreated algorithmically in "pseudo" mode, thereby exhibiting innate vulnerability. Proposing a secure protocol based on unshared randomized data, which therefore can be based on 'white noise' or other real-world, non algorithmic randomization. Prospective use of this T-Proof protocol includes proving possession of data to a party in possession of same data. The principle: Alice wishes to prove to Bob that she is in possession of secret data s, known also to Bob. They agree on a parsing algorithm, dependent on the contents of s, resulting in breaking s into t distinct, consecutive sub-strings (letters). Alice then uses unshared randomization procedure to effect a perfectly random transposition of the t substrings, thereby generating a transposed string s'. She communicates s' to Bob. Bob verifies that s' is a permutation of s based on his parsing of s to the same t substrings, and he is then persuaded that Alice is in possession of s. Because s' was generated via a perfectly randomized transposition of s, a cryptanalyst in possession of s' faces t! s-candidates, each with a probability of 1/t! (what's more: the value of t, and the identity of the t sub-strings is unknown to the cryptanalyst). Brute force cryptanalysis is the fastest theoretical strategy. T-Proof can be played over s, mixed with some agreed upon nonce to defend against replay options. Unlike the competitive solution of hashing, T-Proof does not stand the risk of algorithmic shortcut. Its intractability is credibly appraised.

Introduction

Online connection dialogues normally start by Alice logging on to Bob's website, passing along name, account number, passwords etc. -- data items well possessed by Bob. Such parties normally establish a secure channel beforehand but (i) the secure channel is vulnerable to man-in-the-middle (MiM) attacks, and (ii) at least some such information may be passed along before the secure channel is established (e.g. name, account number). It is very easy for Bob to send Alice a public encryption key, and ask her to encrypt her secret data s with that key, but this solution is also vulnerable to MiM attacks. Hashing is one effective solution, but it relies on the unproven hashing complexity. Here we propose a solution for which "brute force" is the best cryptanalytic strategy: T-Proof (T for transposition): Alice wishes to prove to Bob that she is in possession of a secret, s, known to Bob. Bob sends Alice random data, r, with instructions how to "mix" s and r into q which appears randomized. q is then parsed to t letters according to preset rules. And based on these t letters q is randomly transposed to generate q'. q' is then communicated to Bob over insecure lines. Bob verifies that q' is a permutation of q, and concludes that Alice is in possession of s. A hacker unaware of q will not know how q is parsed to t letters, and hence would not know how to reverse-transpose q' to q. Unlike the prevailing hashing solutions and their kind, T-Proof is not based on algorithmic complexity, rather on solid combinatorics, whereby the user can credibly estimate the adversarial effort to extract the value of the proving secret s. Alice and Bob need to share no secret key to run the T-Proof procedure. T-Proof is computationally easy, operates with any size of secret s, and may be used by Alice to identify to Bob who she is, while keeping her identity secret towards any eavesdropper. It may be used by a group to prove the identities of files, and databases kept by each member of the group. Unlike hashing, T-Proof, in some versions, does not stand the risk of collision, only brute force attack, the required effort of which may be controlled by the user.

The anchor of security online is a "cyber passport" authoritatively and replaceable issued off-line, and then securely used for identification and other purposes. Inherently using an identification code to prove identity is a procedure in which the identity verifier knows what id to expect. Customarily, people and organizations have simply sent their id to the verifier, in the open. More sophisticated means include some form of encryption. Alas, If Alice sends Bob a cipher to encrypt his message to her with it, then this cipher may be confiscated by a hacker in the middle, who will pretend to be Alice when he talks to Bob, and gives him his version of "Alice's cipher", which Bob uses and thereby reveals to the hacker his secret data (id, account number, password, etc). Bob then uses Alice's cipher to send her the same, and Alice is never the wiser.

A more effective solution is one where a stealth man in the middle cannot compromise the proving data. One such method is hashing. Hashing is based on unproven complex algorithms, and collision is always a worry. So it makes sense to come up with alternative means for a party to prove to a verifier aware of s, that the prover is in possession of s.

This proposed solution is based on the idea that the prover may parse her secret bit string s, to some t letters, where a letter is some bit sequence. The procedure to parse s to t letters is a function of s. Then the prover, randomly transposes the t letters, to create an equal length string s'. s' is sent over to the verifier. The verifier, in possession of s will use the same parsing procedure to identify the same t letters in s, and then verify that s' is a strict permutation of s. This will convince the verifier that the prover has s in his or her possession. A hacker, capturing s' will not know what t letters s' is comprised of, and anyway since s' is a random permutation of s, the hacker will not know how to reverse transpose s' to s.

Illustration: The prover, named John Dow, wishes to let the verifier know that he asks to log in. Using T-Proof Mr. Dow will write his name (s) in ASCII:

s = 01001010 01101111 01101000 01101110 00100000 01000100 01101111 01110111

Let's parse s as follows: the first bit is the first letter "A", the next two bits are the second letter, "B", the third letter is comprised of the four next letters, etc:

A=0, B=10, C=0101, D=00110111, E=1011010000110111

F= 00010000001000100011011111011110111

s = 0 10 0101 00110111 1011010000110111 00010000001000100011011111011110111 =ABCDEF

Let's now randomly transpose the t=6 letters (A, B, C, D, E, F) to write:

s' =T(s) = ECFABD = 1011010000110111 0101 00010000001000100011011111011110111 0 10 00110111

, Or:

s' = 10110100 00110111 01010001 00000010 00100011 01111011 10111010 00110111

The verifier, in possession of s, will similarly break s to A,B,C,D,E,F letters, then, starting from the largest letter, F=00010000001000100011011111011110111, the verifier will find the "F-signature" on s':

s'=1011010000110111 0101 F 0 10 00110111

then the "E-signature": E=1011010000110111

s'=E 0101 F 0 10 00110111

And so on to construct s'=ECFABD. The verifier will conclude then that s' is a perfect permutation of s, based on the six letters A, B, C, D, E, F. All letters were found in s', and no unmarked bit left in s'.

If the verifier does not know the name John Dow, then the verifier will list all the names in its database pre-parsed by their proper letters, and compare s' to this expression of the names.

The hacker, capturing s' cannot parse it to the proper letters (A, B, C, D, E, F) because, unlike the verifier, the hacker does not know s. If the hacker uses the same parsing rules on s', he gets: A'=1, B'=01, C'=1010, D'=00011011, E'=1010100010000001, F'=000100011011101110111010. So clearly: A' ≠ A, B' ≠ B, C' ≠ C, D' ≠ D, E' ≠ E, F' ≠ F. So s' cannot be interpreted by the hacker as a permutation of s, except after applying the prolonged brute force cryptanalysis.

Notice that the verifier and the prover need not share any secrets to collaborate on this T-Proof procedure. They just need to adhere to this public protocol.

There are many variations on this procedure to balance security and convenience, but this illustration highlights the principle.

The T-Proof Environment

The environment where T-Proof operates is as follows: three parties are involved: a prover, a verifier, and a hacker. A measure of data regarded as secret s is known to the prover and to the verifier, and not known to the Hacker. The prover and the verifier communicate over insecure lines with the aim of convincing the verifier that the prover is in possession of s -- while making it hard for the Hacker to learn the identity of s. The

verifier and the prover have no shared cryptographic keys, no confidential information. They both agree to abide by a public domain protocol.

T-Proof is a public function that maps s to s', such that by sending s' to the verifier, the prover convinces the verifier that the prover is in possession of s, while the identity of s', assumed captured by the hacker, makes it sufficiently intractable for the Hacker to infer s.

We are interested in the following probabilities: (1) the probability for the verifier to falsely conclude that the prover holds s, and (2) the probability for the Hacker to divine s from s'. We rate a solution like T-Proof with respect to these two probabilities.

The T-Proof Principles

The T-Proof principle is as follows: let s be an arbitrary bit string of size n: $s= s_0 =\{0,1\}^n$. Let s be parsed into t consecutive sub-strings: $s_1, s_2, ...s_t$, so that:

$$s_0 = s_1 s_2 ..... s_t$$

Let s' be a permutation of s based on these t substrings. Any one in possession of s, will be able to assert that s' is a permutation of s (based on the t sub-strings), and will also be able to compute the number of possible s-string candidates that could have produced s' as their permutation. Based on this number (compared to 2^n) one will be able to rate the probability that s' is a permutation of some s" ≠ s. Given that the string s is highly randomized (high entropy), then anyone in possession of s' but without the possession of s, will face well defined set of randomized possibilities for the value of t and for the sizes of $s_1, s_2,....s_t$ such that by some order, o, these substring will construct s':

$$s'_o = s_i s_j s_k ... s_t .....$$

T-Proof is then a method for a prover to prove that she has a measure of data s, known to the verifier, such that it would be difficult for a Hacker to infer the value of s, and where both the probabilities for verifier error and for Hacker's success are computable with solid durable combinatorics, and the results are not dependent on assumed algorithmic complexity.

Auxiliary principles: (a) to the extent that s is a low entropy string, then it may be randomized before submitting it to T-proof. For example encrypting s with any typical highly randomizing cipher. The cipher key will be passed in the open since what is needed here is only the randomization attribute of the cipher, not its secrecy protection. (b) In order for the prover to be able to prove possession of same s time and again (in subsequent sessions), she might want to "mix" s with a random bit sequence r, to generate a new string, q, and apply T-Proof over q.

T-Proof Design

The T-Proof procedure is comprised of the following elements:

- **Non-Repetition Module**
- **Entropy Enhancement Module**
- **Parsing Module**
- **Transposition Module**
- **Communication Module**
- **Verification Module**

These modules operate in the above sequence: the output of one is the input of the next.

Non-Repetition Module

In many cases the prover would wish to prove the possession of s to the verifier in more than one instant. To prevent a hacker from using the "replay" strategy and fool the

verifier, the prover may take steps to insure that each proving session will be conducted with new, previously unused, and unpredictable data.

One way to accomplish this is to "mix" s with a nonce, a random data, r, creating q=mix(s,r). The mixing formula will be openly agreed upon between the prover and the verifier. The "mix" function may be reversible, or irreversible (lossy or not lossy). Namely given q and r it may be impossible to determine the value of s, since many s candidates exist, or, alternatively, given r and q, s will be determinable. It will then be a matter of design whether to make it intractable to determine s from r and q, or easy.

One consideration for r and the "mix" is the target bit size of the value that undergoes the T-Proof procedure. That size can be determined by selecting r and ‘mix’.

Since the procedure computed by the prover will have to also be computed by the verifier, (except the transposition itself), it is necessary that r will have to be communicated between the two. Since the verifier is the one who needs to make it as difficult as possible for the prover to cheat, it makes more sense for the verifier to determine r, (different per each session), and pass it on to the prover. The mix function, too, may be the purview of the verifier.

The simplest mix option is concatenation of s with r: q=sr, and r is adjusted to get the right size q.

Entropy Enhancement Module

Once the secret s is preprocessed to become q (the non-repetition module), it may be advisable to pump in entropy to make it more difficult for the hacker to extract the secret (s or q). Linguistic data (name, addresses) are of relatively low entropy, and can be better

guessed than purely randomized data. It is therefore helpful for the users to "randomize" q. The randomization process, also will be in the open, and known to the hacker.

An easy way to randomize q is to encrypt it with a public key using any established cipher.

Parsing Module

Given a string s comprised of n bits: $s=s_0=\{0.1\}^n$, it is possible to parse it to t consecutive substrings $s_1s_2.....s_t$, where $1 \leq t \leq n$. Based on these t substrings s may be transposed up to t! permutations. So for every secret s, there are at most t! s' candidates. Or, alternatively, given s' the hacker will face up to t! s-candidates. Therefore, it would seem that one should try to maximize t.

The hacker facing the n-bits long s' string does not know how the sub-strings are constructed. The hacker may or may not know the value of t. Clearly if t=1 then s'=s. If t=2, then the cut between the two substrings may be from bit 2 to bit n-1 in s'. If the substrings are all of equal size then their identity is clear in s'. If the hacker is not aware of t or of any substring size (because it depends on s, which is unknown to him), then given s' the hacker will face a chance to guess s:

$$Pr[x=s] = 1/C^{t-1}_{n-2}$$

where x is any s candidate, and C^{t-1}_{n-2} is the number of ways that (t-1) split points can be marked on the n bits long string. This guessing probability decreases as t increases (and the substrings decrease).

On the other hand, larger t would make it more difficult for the verifier to check whether s' is a permutation of s based on the parsed substrings. A large t, implies small

sub-strings. A small sub-string of an average size of (n/t) bits will probably fit on different spots on s', and the verifier would not know which is the right spot.

Illustration: Let s' = 10101110101000101110. for a substring s_i=101 the verifier will identify 5 locations to place it on s'. And or s_j = 111, there are two locations. By, contrast a larger substring s_k = 1000101 will fit only in one location on s'.

One would therefore try to optimize the value of t and the various sub-string sizes between these two competing interests.

Some design options are presented ahead:

- ***The Incremental Strategy***
- ***The Minimum size strategy***
- ***The log(n) strategy***

These strategies are a matter of choice, each with its pro and cons.

We keep here the s, s' notation, but it should also apply to instances where the "entropy enhancement" module is applied, and then s, and s' will be replaced by q and q'.

The Incremental Strategy

The "minimum size strategy" works as follows: s is approached from left to right (or alternatively, from right to left). The first bit is regarded as the first letter, let's designate it as A. A is either "1" or "0". Then one examines the second bit. If it is different from the first bit then it is set as B. If the second bit is of the same value as the first bit, then the next bit is added, and the two-bit string becomes B. Further, one examines the next two bits, if they look the same as a previous letter, one moves up to three bits, and so on. When the last letter so far was defined as l bits long, and there are only $m \leq 2l$ letters left in s, then the last letter is extended to include these m bits.

This strategy increments the size of the letters, and the parsing of the string s depends on the bit value of s. And hence, knowing only s', the hacker will not know how s was parsed out, not even the value of t -- the number of sub-strings. As designed s is parsed into t non-repeat letters, and hence s will have t! permutations.

This strategy can be modified by starting with bit size of $l > 1$, and incrementing "+2" or more instead of "+1" each round.

There might rise a slight difficulty for the verifier looking at s' trying to verify that s substrings fit into s'.

Illustration (incremental strategy)

The prover, Bob, wishes to convince the verifier, Alice, that he has in his possession Bob's PIN, which is: $s=8253_{10} = 10000000111101$

Bob then decomposes s to a sequence of non-repeat letters, from left to right, starting with a bit size letter: The first leftmost bit is 1, so Bob marks a=1. The next bit is zero, Bob marks b=0 ($a \neq b$). The third bit is a zero too, so it would not qualify for the next letter. Bob then increments the size of the letter to two bits, and writes c=00. ($c \neq b \neq a$). What is left from s now is:

s=~~1000~~0000111101

The next 2 bits will not qualify as d, since then we have d=c, which Bob wishes to avoid, so Bob once again increases the bit count, now to three and writes d=000 ($\neq c \neq b \neq a$). s now looks like:

s=~~1000000~~0111101

The next three bits will qualify as e=011, because $e \neq d \neq c \neq b \neq a$), and the same for f=110 $\neq e \neq d \neq c \neq b \neq a$. Now:

$$s=\cancel{10000000111101}$$

One bit is left unparsed it could not be g=1 since then g=a, so the rule is that the left over bits are concatenated to the former letter, hence we rewrite: f=1101

At this point we can write:

$$s = abcdef$$

where the 6 letters that comprise s are defined above.

Bob will then randomly transpose s per these 6 letters and compute an s-transpose:

$$s' = dbfeac$$

Bob will now transmit s' to Alice using its binary representation:

$$s' = 000\ 0\ 1101\ 011\ 1\ 00$$

But not with these spaces that identify the letters, rather:

$$s' = 00001101011100 = 860$$

Alice receiving s', and having computed the letters in s, like Bob did (Alice is in possession of s), will now check whether the s' that Bob transmitted is letter-permutation of s (which she computed too).

To do that Alice starts with the longest letter: f=1101, and moves it from the rightmost bits in s':

$$s' = 0000\ [1101]_f\ 011100$$

Alice will then look if e=011 fits in s':

$$s' = 0000\ [1101]_f\ [011]_e\ 100$$

Continuing with d=000:

$$s' = 0\ [000]_d\ [1101]_f\ [011]_e\ 100$$

And so on, until Alice, the verifier, securely concludes that s' is a permutation of s based on the incremental parsing strategy of s.

The Minimum Size Strategy

This strategy is similar to the incremental size strategy. The difference is that one tries to assign minimum size for each next sub-string.

Regarding the former illustration, let $s=8253_{10}$ = 10000000111101. It will be parsed a=1, b=0, c=00, d=000, resulting in s= ~~1000000~~0111101. But the next letter, will be e=01, because there is no such letter so far. And then f=11. We now have: s= ~~10000000111~~101. The next letter could have been g=10 because this combination was not used before. But because only 1 bit is left in s, we have g=101. Clearly the parsing of s is different by the two strategies, even the number of sub-strings (letters) is different.

The log(n) Strategy

This strategy is one where matching s' to the sub-strings of s is very easy. But unlike the former two strategies, the parsing of s (comprised of n=|s| bits) is by pre-established order, independent of the contents of s.

Procedure: Let L^j_i be letter i (or, say sub-string i) from the j series alphabet. For every letter series j we define, the size of the letters:

$$| L^j_i | = 2^i$$

Accordingly one will parse a bit string s as follows:

$$s = L^j_1 L^j_2 ..... L'^j_t$$

where L'^j_t has the length $l = |s| - (2^0 + 2^1 + 2^2 + .. 2^{t-1})$, where t is the smallest integer such that $|s| \leq 2^t$. Accordingly $t \sim \log_2(|s|) = \log_2(n)$.

Illustration: Let s = 1 01 0010 00100001 0000010000001, we parse it as follows:
$L^1_0 = 1$, $L^1_1 = 01$, $L^1_2 = 0010$, $L^1_3 = 00100001$, $L'^1_4 = 0000010000001$

Security and convenience considerations may indicate that the last letter L'^1 is too large. In that case it will be parsed according to the same rules, only that its sub-strings will be regarded as a second letters sequence:

$$L'^1_t = L^2_0 L^2_1 L^2_2 .... L'^2_{t'}$$

Note that for every round of log(n) parsing there would be exactly one possible position for every substring within s', because every sub-strings is longer than all the shorter substrings combined. This implies a very fast verification process.

Illustration, the last letter above: $L'^1_4 = 0000010000001$ may be parsed into: $L^2_0 = 0$, $L^2_1 = 00$, $L^2_2 = 0010$, $L^2_3 = 0000001$

The last letter in this sequence can be parsed again, and so on, as many times as one desires. The log(n) strategy might call for all sub-strings of size 2^m and above to be re-parsed.

The verifier, knowing s will be able to identify all the letters in the parsing. And then the verifier will work its way backwards, starting from the sub-string that was parsed out last. The verifier will verify that that letter is expressed in some order of its due sub-

strings, and then climb back to the former round until the verifier verifies that s' is a correct permutation of the original s string.

This strategy defines the parsing of every bit string, s, regardless of size. And the longer s, the greater the assurance that the prover indeed is in possession of s.

The Smallest Equal Size Strategy

This strategy parses s to (t-1) equal size sub-strings (letters), and a t letter of larger size. One evaluates the smallest letter size such that there is no repeat of any letter within s.

Given a bit string s, $\{0,1\}^n$, for l=1 one marks m l bits long substrings starting from an arbitrary side of s (say, leftmost) where m = (n - n mod l) l. These leaves u = n - l*m bits unmarked (u < l). If any two among these m substrings are identical, then one increments l, and tries again iteratively until for some l value all the m substrings are distinct. In the worst case it happens for an even n at l=0.5*n+1, and for an odd n at l=0.5(n+1). Once the qualified l is identified, the first (m-1) substrings are declared as the first (t-1) substrings of s, and the m-th l bits long substring is concatenated with the remaining u bits to form a l+u bits long substring. The thus defined t substrings are all distinct, and it would be very easy for the verifier to ascertain that s' is a t-based permutation of s. On the other hand, the hacker will readily find out the value of t because applying this procedure to s' will likely result in the same value of t. So the only intractability faced by the hacker would be the t! size permutation space.

Illustration: let s=10010011101001110. For l=1 we have several substrings that are identical to each other. Same for l=2. We try then for l=3:

s=100 100 111 010 011 10

There are two identical strings here, so we increment *l*=4:

s=1001 0011 1010 0111 0

Now, all the four, four bit size substrings are distinct, s is parsed into: 1001,0011,1010,01110.

Transposition Module

The T-Proof transposition should be randomized to deny the hacker any information regarding reversal, so that given s' the hacker will face all t! possible permutation, each with a probability of 1/t!. This can be done based on the "Ultimate Transposition Cipher [7], or by any other methods of randomization. It is important to note that the randomization key is not communicated by the prover to the verifier, so the prover is free to choose and not communicate it further.

One simple example for randomized permutation is as follows: the string s is comprised of t sub-strings: s_1, s_2, ... s_t. When substring s_i is found in position j in the permutation s', then we shall designate this string as s_{ij}.

Using repeatedly a pseudo random number generator, the prover will randomly pick two numbers $1 \leq i \leq t$, and $1 \leq j \leq t$, and so identify s_{ij}. Same will be repeated. If the random pick repeats a number used before (namely re-picks the same i, or the same j), then this picking is dropped, and the random number generator tries again. This randomization process is getting slower as it progresses.

Another variety is to pick the next unused index (i, and j) if a used value is re-selected.

Communication Module

The communication module needs to submit s' and some meta data describing the protocol under which the string s' is being sent.

The module might have also to communicate the random nonce to the prover, and the confirmation of the reception of the s information.

Verification Module

Let's first develop the verification procedure for a simple permutation, s' (as opposed to the several rounds of transposition as in the log(n) strategy). Procedure: the verifier first tries to fit the longest substring into s' (or one of the longest, if there are a few). If there is no fit, namely, there is no substring on s' that fits the longest substring checked, then the verification fails. If there is one fit, then the fitted bits on s' are marked as accounted for. The verifier then takes the next largest substring and tries to fit it somewhere in the remaining unaccounted bits of s'. If no fit -- the verification fails. If there is a single fit, the above process continues with the next largest substring. This goes on until the verification either fails, or concludes when all the substrings are well fitted into s' and the verifier then ascertains that there are no left-over unaccounted for bits. If there are leftover bits -- the verification fails.

If for any substring there are more than one places of fit, then, one such place is chosen, and the other is marked for possible return. The process continues with the picked location. If the verification fails at some point, the verifier returns to the marked alternative, and continues from there. This is repeated at any stage, and only if all possible fittings were exhaustively checked and no fit was found, then the verification as a whole fails. If somewhere along the process a fit is found then the verification succeeds.

In the case of several rounds as in the log(n) parsing strategy, then the above procedure is repeated for each round, starting from the last parsing.

Different parsing strategies lead to different efficiencies in verification.

Applications

T-Proof may be applied in a flexible way to provide credibly estimated security to transmission of data already known to the recipient. The most natural application may be the task of proving identity and possession of identity-related data, but it is also a means to insure integrity and consistency of documents, files, even databases between two or more repositories of the same.

Proving Identity

When two online entities claim to be known to each other and hence start a dialogue, then the two may first identify themselves to each other via T-Proof. In particular, if Alice runs an operation with subscribers identified by secret personal identification numbers, PIN, then Bob, a subscriber, may use T-Proof to prove his identity to Alice, and in parallel Alice, will use T-Proof to prove to Bob that she is Alice, and not a fishing scheme. In that case they may each apply the entropy enhancement module with the other supplying the necessary randomness.

Alice could store the PINs or names, etc. with their parsed letters so that she can readily identify Bob although he identifies himself through T-Proof.

Proving Possession of Digital Money

Some digital money products are based on randomized bit strings (e.g. BitMint). Such digital coins may be communicated to an authentication authority holding an image of this coin. T-Proof will be a good fit for this task.

Acceptable Knowledge Leakage Procedures

Alice may wish to prove to Bob her possession of a secret s, which Bob is not aware of. So Bob passes Alice communication to Carla, who is aware of s, and he wishes Carla to confirm Alice's claim that she is in possession of s. By insisting on going through him, Bob is assured that Carla confirms the right s, and also it gives him the opportunity to test Carla by forwarding some data in error. Alice, on her part, wishes to prevent Bob from subsequently claiming that he knows s. She might do so over a randomized s, by extracting from s some h bits, and constructing an h bits long string over which Alice would practice T-Proof. h should be sufficiently large to give credibility to Carla's confirmation, and on the other hand is should be a sufficiently small fraction of s, to prevent Bob form guessing the remaining bits.

Cryptanalysis

Exact cryptanalysis may only be carried out over a well defined set of parameters of a T-Proof cipher. In general terms though, one can assert that for well randomized pre-transposition data (randomized q) there is no more efficient way than brute force. Proof: The hacker in possession of s', trying to deduce s, will generally not know how s' is parsed out: often not to how many substrings, and mostly not the size and not the identity of these substrings. But let us, for argument's sake, assume that the t substrings have all somehow became known to the Hacker. Alas, what was never communicated to the verifier is the transposition key from s to s'. What is more, this transposition was carried out via a randomized process, and hence given s', there are t! s-candidates, and each of them is associated with a chance of 1/t! to be the right s. There is no algorithm to crack, or to shortcut, only the randomization process underlying the transposition. To the extent that an algorithmic pseudo-random process is used, it can be theoretically cryptanalyzed. To the extent that a randomized phenomenon is used, (e.g. electronic white noise) it can't be cryptanalyzed. Since the prover does not communicate the transposition key, or

formula, and does not share it with anyone, the hacker faces a de-facto proper randomization, and is left with only brute force as a viable cryptanalytic strategy.

In general one must assume built-in equivocation, namely given s' there may be more than one s-candidates that cannot be ruled out by the cryptanalyst. Such equivocation may be readily defeated by running two distinct entropy enhancement modules, to produce two distinct permutations s'_1, s'_2.

Unlike hashing, which is an alternative solution to the same challenge, T-Proof is getting more and more robust for larger and larger treated data. The user will determine the level of security over say a large file, or database, by deciding how to break it up to smaller sections, and apply T-Proof to each section separately. It is easier and faster to apply to smaller amounts of data, but security is less.

Reference

1. Samid, G. "Re-dividing Complexity between Algorithms and Keys" Progress in Cryptology — INDOCRYPT 2001 Volume 2247 of the series Lecture Notes in Computer Science pp 330-338

2. Samid, G. 2001 "Anonymity Management: A Blue Print For Newfound Privacy" The Second International Workshop on Information Security Applications (WISA 2001), Seoul, Korea, September 13-14, 2001 (Best Paper Award).

3. Samid, G. 2001 "Re-Dividing Complexity Between Algorithms and Keys (Key Scripts)" The Second International Conference on Cryptology in India, Indian Institute of Technology, Madras, Chennai, India. December 2001.

4. Samid, G. 2003 "Intractability Erosion: The Everpresent Threat for Secure Communication" The 7th World Multi-Conference on Systemics, Cybernetics and Informatics (SCI 2003), July 2003.

5. Samid, (A) 2015 “Equivoe-T: Transposition Equivocation Cryptography” 27 May 2015 International Association of Cryptology Research, ePrint Archive https://eprint.iacr.org/2015/510

6. Ma´t´e Horva´th, 2015 “Survey on Cryptographic Obfuscation” 9 Oct 2015 International Association of Cryptology Research, ePrint Archive https://eprint.iacr.org/2015/412

7. Samid, 2015 “The Ultimate Transposition Cipher (UTC)” 23 Oct 2015 International Association of Cryptology Research, ePrint Archive https://eprint.iacr.org/2015/1033